MAC 0331

Geometria Computacional

Pedro Gigeck Freire

10737136

11/04/2020

Lista 3

1] Seja S um conjunto de n segmentos de retas disjuntos cujo ponto extremo superior pertence a reta $y=1$ e o ponto inferior pertence a reta $y=0$

Estes segmentos particionam a faixa horizontal $]-\infty, \infty[\times [0,1]$ em $n+1$ regiões

Descreva um algoritmo que constrói uma árvore de busca binária com os segmentos de S tal que a região contendo um dado ponto possa ser determinada em tempo $O(\lg n)$.

A etapa de construção deve consumir tempo $O(n \lg n)$. Também descreva o algoritmo de busca em detalhes.

Construção da Árvore:

Primeiramente, devemos ordenar os segmentos, em ordem da esquerda pra direita. Como os segmentos são disjuntos, podemos ordenar em ordem crescente da coordenada x do ponto superior de cada segmento.

Com o S ordenado, podemos criar nossa ABB a partir de uma busca binária tradicional: a raiz é o segmento do meio e as subárvores podem ser feitas recursivamente

```
ConstroiABB(S, n):
1    MergeSort(S, n)
2    devolva ConstroiABBrec(S, 1, n)
```

```
ConstroiABBrec (S, l, r):
1     se  r - l < 0
2         então devolva NIL
3     meio ← ⌊(r + l) / 2⌋
4     raiz.seg ← S[meio]
5     raiz.esq ← constroiABBrec(S, l, meio-1)
6     raiz.dir ← constroiABBrec(S, meio+1, r)
7     devolva raiz
```

Sabemos que essa recorrência consome tempo linear, então nosso ConstroiABB() consome tempo $O(n \lg n)$ por causa da ordenação.

Sabemos também que a ABB ficou balanceada, pois as subárvores de cada nó tem o mesmo número de nós (+ ou - 1), então a altura da árvore é $O(\lg n)$.

Para determinar qual região contém algum ponto, devemos percorrer a árvore usando o predicado esquerda()

É importante que os segmentos estejam todos orientados no mesmo sentido (de baixo pra cima), podemos preprocessar isso durante a ordenação, por exemplo.

```
Busca (raiz, p):          ▷ Função que recebe um ponto p a raiz da ABB
1     atual ← raiz
2     anterior ← NIL
3     enquanto atual ≠ NIL faça
4         anterior ← atual
5         se esquerda (atual.seg, p)
6             atual ← atual.esq
7         se não
8             atual ← atual.DIR
9     se esquerda (anterior.seg, p):
10        devolva {predecessor(anterior).seg, anterior.seg}
11    se não
12        devolva {anterior.seg, sucessor(anterior).seg}
```

Nossa função de busca faz uma busca binária na prática.

Como antecessor() e sucessor() consomem tempo $O(n \lg n)$ então buscamos um ponto p em $O(n \lg n)$.

4] Mostre um algoritmo que, dados dois Polígonos [redacted] simples P_1 e P_2, com um total de n vértices, decide se P_1 e P_2 se intersectam. Seu algoritmo deve consumir tempo $O(n \lg n)$.

O Algoritmo que vimos em aula para interseção de segmentos com linha de varredura pode realizar o serviço, com algumas modificações.

O mais importante é não detectar interseção entre arestas do mesmo polígono, então devemos identificar cada segmento com seu polígono.

Ignorar as interseções entre os segmentos do mesmo polígono não quebraria nosso algoritmo, pelo fato dos polígonos serem simples, ou seja, não precisaríamos alterar a linha de varredura pois quando os segmentos se intersectam concomitantemente eles terminam. (nos vértices)

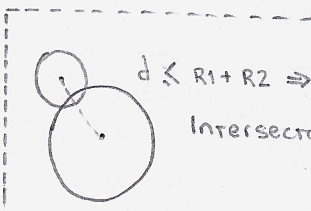
Com isso, teríamos n segmentos para rodar no nosso algoritmo $O(n \lg n)$

5] Um disco é a região do plano limitada por uma circunferência. Um disco é dado através das coordenadas do seu centro e o valor do seu raio. Dois discos se intersectam se eles tem algum ponto em comum. Descreva um algoritmo que determine se há dois discos que se intersetam na coleção em tempo $O(n \lg n)$

Vamos usar, também, a técnica de linha de varredura

Primeiramente, dois discos se intersectam se a soma de seus raios for maior que a distância entre seus centros

```
INTERSECTA (D1, D2)
1  DEVOLVA [DIST2(D1.CENTRO, D2.CENTRO) ≤ (D1.RAIO + D2.RAIO)^2]
```



Nossa linha de varredura será vertical e varrerá da esquerda para direita.

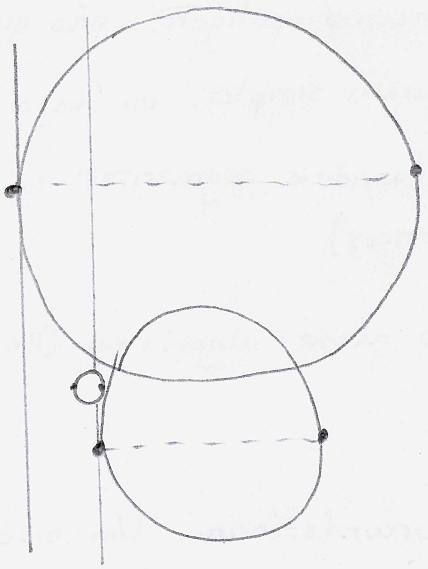
E a ideia também será igual ao de algoritmo de interseção de segmentos, a linha guardará os discos que intersectam a linha, e usaremos de referencia uma "linha do Equador" de cada disco.

O algoritmo:

* {
- Trasforma os discos em segmentos (As linhas do Equador)
- Guarda junto de cada segmento o disco que ele representa

- Roda o algoritmo de detectar interseção de segmentos, usando o teste de interseção descrito anteriormente.

O que garante que algoritmo funciona é que se dois discos se intersectam então os seus 'segmentos equatoriais' ficam consecutivos na linha de varredura



Se eles não ficam consecutivos, então há um disco entre eles.

- Uma hora esse disco vai acabar e nossos discos ficarão consecutivos

(Se o disco do meio não acabar, então ele também intersecta).

O passo * consome $O(n)$, então consumimos os mesmos $O(n \lg n)$ que o algoritmo de interseção de segmentos. ☺